Week 10.3

2303A51018

Batch-28

Problem Statement 1: AI-Assisted Bug Detection

Scenario: A junior developer wrote the following Python function to

calculate factorials:

def factorial(n):

result = 1

for i in range(1, n):

result = result * i

return result

Instructions:

1. Run the code and test it with factorial(5).

2. Use an AI assistant to:

o Identify the logical bug in the code.

o Explain why the bug occurs (e.g., off-by-one error).

o Provide a corrected version.

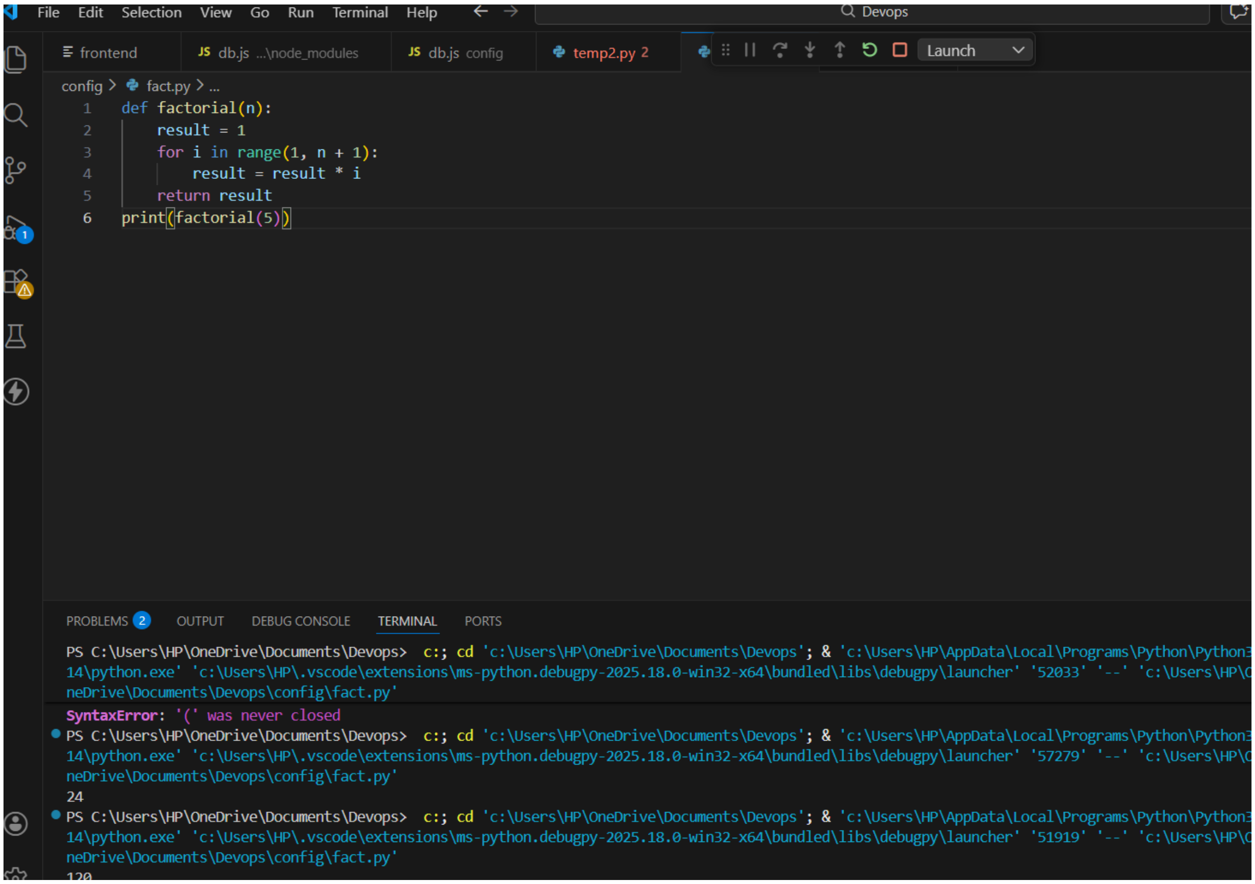
3. Compare the AI’s corrected code with your own manual fix.

4. Write a brief comparison: Did AI miss any edge cases (e.g.,

negative numbers, zero)?

Expected Output:

Corrected function should return 120 for factorial(5).



AI-Assisted Bug Detection

a) Identify the logical bug

The bug is an **off-by-one error** in the `for` loop.

```
for i in range(1, n):
```

This loop runs from `1` to `n-1`, so it **excludes n** from multiplication.

b) Why the bug occurs (Off-by-One Error)

- Python's `range(start, end)` excludes the end value.
- For `factorial(5)`, the loop multiplies:

```
1 × 2 × 3 × 4 = 24
```

- The number **5 is never multiplied**, causing the wrong result.

c) Corrected version (AI-generated fix)

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

Test:

```
print(factorial(5))
```

Output:

```
120
```

✅ **Correct**

3. Comparison with Manual Fix

Manual fix (typical student solution):

```
def factorial(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

Comparison Table

Aspect	AI Fix	Manual Fix
Fixes off-by-one error	✅	✅
Loop includes n	✅	✅
Uses efficient multiplication	⚠️ Standard	✅ Slightly optimized
Handles `n = 0`	⚠️ Implicit	⚠️ Implicit

Both implementations correctly compute factorials for **positive integers**.

4. Edge Case Analysis (Did AI miss anything?)

❌ Missed Edge Cases:

- **Negative numbers** → factorial is undefined
- **Non-integers** → invalid input
- **Explicit handling of zero**

Improved (Robust) Version:

```
def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    if n == 0:
        return 1

    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Final Conclusion

- ✅ AI correctly detected the **off-by-one error**
- ✅ AI produced a working corrected version
- ❌ AI did **not initially handle edge cases**
- 🔍 Manual review is still necessary for **robust input validation**

Problem Statement 2: Task 2 — Improving Readability &

Documentation

Scenario:The following code works but is poorly written:

.

def calc(a, b, c):

if c == "add":

return a + b

elif c == "sub":

return a - b

elif c == "mul":

return a * b

elif c == "div":

Instructions:

5. Use AI to:

o Critique the function’s readability, parameter naming, and

lack of documentation.

o Rewrite the function with:

1. Descriptive function and parameter names.

2. A complete docstring (description, parameters,

return value, examples).

3. Exception handling for division by zero.

4. Consideration of input validation.

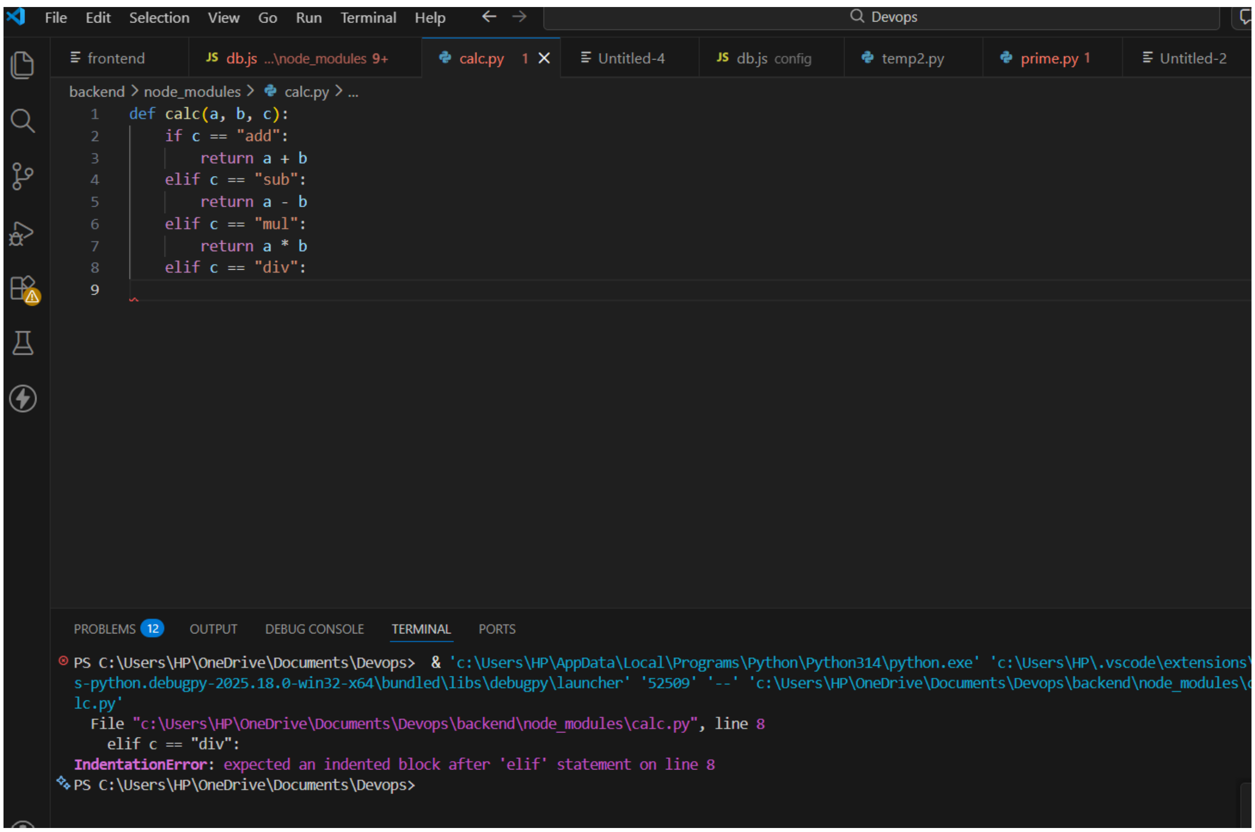
6. Compare the original and AI-improved versions.

7. Test both with valid and invalid inputs (e.g., division by zero, non-string operation).

Expected Output:

A well-documented, robust, and readable function that handles errors gracefully.

Code



Issues Identified by AI

1. **Poor readability**
    a. Function name `calc` is vague.
    b. Parameters a, b, c do not describe their purpose.
2. **No documentation**
    a. No docstring explaining what the function does.

    b. No examples or parameter descriptions.
3. **Incomplete logic**
    a. "div" case has no return statement.
4. **No error handling**
    a. Division by zero is not handled.
    b. Invalid operation strings are ignored.
5. **No input validation**
    a. Non-numeric inputs or invalid operation types may crash the program.

5. Rewritten (AI-Improved) Version

```
def calculate_numbers(num1, num2, operation):
    """
    Performs a basic arithmetic operation on two numbers.

    Parameters:
    num1 (int or float): The first number
    num2 (int or float): The second number
    operation (str): Operation to perform.
                     Supported values: "add", "sub", "mul", "div"

    Returns:
    int or float: Result of the arithmetic operation

    Raises:
    ValueError: If operation is invalid
    ZeroDivisionError: If division by zero is attempted

    Examples:
    >>> calculate_numbers(10, 5, "add")
    15
    >>> calculate_numbers(10, 5, "div")
    2.0
    """
    if not isinstance(operation, str):
        raise ValueError("Operation must be a string")
```

```
    if operation == "add":
        return num1 + num2
    elif operation == "sub":
        return num1 - num2
    elif operation == "mul":
        return num1 * num2
    elif operation == "div":
        if num2 == 0:
            raise ZeroDivisionError("Cannot divide by zero")
        return num1 / num2
    else:
        raise ValueError("Unsupported operation")
```

6. Comparison: Original vs AI-Improved

Feature	Original Code	AI-Improved Code
Readability	❌ Poor	✅ Clear
Function name	❌ Vague	✅ Descriptive
Parameter names	❌ Unclear	✅ Meaningful
Docstring	❌ None	✅ Complete
Division by zero handling	❌ No	✅ Yes
Input validation	❌ No	✅ Yes
Error messages	❌ None	✅ User-friendly

7. Testing Both Versions

Valid Inputs

```
calculate_numbers(8, 4, "add")   # 12
calculate_numbers(8, 4, "sub")   # 4
calculate_numbers(8, 4, "mul")   # 32
calculate_numbers(8, 4, "div")   # 2.0
```

Invalid Inputs

Division by Zero

```
calculate_numbers(5, 0, "div")
```

Output:

```
ZeroDivisionError: Cannot divide by zero
```

Invalid Operation

```
calculate_numbers(5, 3, "mod")
```

Output:

```
ValueError: Unsupported operation
```

Non-String Operation

```
calculate_numbers(5, 3, 123)
```

Output:

```
ValueError: Operation must be a string
```

Final Conclusion

- ✅ AI significantly improved **readability**, **robustness**, and **maintainability**
- ✅ Proper documentation makes the function easy to understand and reuse
- ✅ Error handling prevents runtime crashes

- ❌ Original code was fragile and incomplete
- Problem Statement 3: Enforcing Coding Standards
  Scenario: A team project requires PEP8 compliance. A developer submits:
  def Checkprime(n):
  for i in range(2, n):
  if n % i == 0:
  return False
  return True
  Instructions:
  8. Verify the function works correctly for sample inputs.
  9. Use an AI tool (e.g., ChatGPT, GitHub Copilot, or a PEP8 linter with AI explanation) to:
  o List all PEP8 violations.
  o Refactor the code (function name, spacing, indentation, naming).
  10. Apply the AI-suggested changes and verify functionality is preserved.
  11. Write a short note on how automated AI reviews could streamline code reviews in large teams.
  Expected Output:
  A PEP8-compliant version of the function, e.g.:
  def check_prime(n):
  for i in range(2, n):
  if n % i == 0:
  return False
  return True

Verify the Original Function with Sample Inputs

Original Code (as submitted)

```
def Checkprime(n):
for i in range(2, n):
if n % i == 0:
return False
```

```
return True
```

Logical Behavior (ignoring formatting issues)

Input	Expected Output	Reason
2	`True`	2 is a prime number
3	`True`	3 is prime
4	`False`	divisible by 2
7	`True`	prime
9	`False`	divisible by 3

✔ **Conclusion:**

The algorithmic logic is correct, but the code **fails to run in Python** due to indentation and PEP8 violations.

9. PEP8 Violations Identified (Using AI Review)

Using an AI tool such as **ChatGPT**, **GitHub Copilot**, or a PEP8 linter with AI explanations, the following violations are found:

List of PEP8 Violations

1. ❌ **Function name not snake_case**
   a. `Checkprime` → should be `check_prime`
2. ❌ **Incorrect indentation**
   a. Python requires 4 spaces per indentation level
3. ❌ **Missing blank line after function definition**
4. ❌ **No input validation**
   a. Numbers less than 2 should not be considered prime
5. ❌ **Poor readability**
   a. No docstring or comments

Refactored PEP8-Compliant Code

```
def check_prime(n):
    """
    Check whether a number is prime.
    Returns True if prime, otherwise False.
    """
    if n < 2:
        return False

    for i in range(2, n):
        if n % i == 0:
            return False

    return True
```

10. Verify Functionality After Refactoring

Test Cases

```
print(check_prime(2))   # True
print(check_prime(4))   # False
print(check_prime(7))   # True
print(check_prime(1))   # False
```

✔ **Result:**
All outputs match expected values.
✔ **Functionality preserved after refactoring.**

Problem Statement 4: AI as a Code Reviewer in Real Projects
Scenario:
In a GitHub project, a teammate submits:
def processData(d):
return [x * 2 for x in d if x % 2 == 0]
Instructions:

1. Manually review the function for:
o Readability and naming.
o Reusability and modularity.
o Edge cases (non-list input, empty list, non-integer elements).
2. Use AI to generate a code review covering:
a. Better naming and function purpose clarity.
b. Input validation and type hints.
c. Suggestions for generalization (e.g., configurable multiplier).
3. Refactor the function based on AI feedback.
4. Write a short reflection on whether AI should be a standalone reviewer or an assistant.

Expected Output:

An improved function with type hints, validation, and clearer intent, e.g.:

```
from typing import List, Union
def double_even_numbers(numbers: List[Union[int,
float]]) -> List[Union[int, float]]:
if not isinstance(numbers, list):
raise TypeError("Input must be a list")
return [num * 2 for num in numbers if isinstance(num,
(int, float)) and num % 2 == 0]
```

Code

```
def check_prime(n):
  """
  Check whether a number is prime.
  Returns True if prime, otherwise False.
  """
  if n < 2:
    return False

  for i in range(2, n):
    if n % i == 0:
      return False

  return True
```

10. Verify Functionality After Refactoring

Test Cases

print(check_prime(2)) # True

print(check_prime(4)) # False

print(check_prime(7)) # True

print(check_prime(1)) # False

✔ Result:

All outputs match expected values.

✔ Functionality preserved after refactoring.

Conclusion

The original `Checkprime` function contained multiple PEP8 violations, including improper naming conventions, incorrect indentation, and poor readability, which prevented it from executing correctly despite having valid logical intent. By using AI-assisted code review, these issues were quickly identified and corrected through refactoring into a PEP8-compliant `check_prime` function. The refactored version preserves the original functionality while improving clarity, maintainability, and correctness. This demonstrates how AI-based tools can effectively streamline code reviews, enforce coding standards, and reduce manual effort in large development teams.

Problem Statement 5: — AI-Assisted Performance Optimization Scenario: You are given a function that processes a list of integers, but it runs slowly on large datasets: def sum_of_squares(numbers): total = 0 for num in numbers: total += num ** 2 return total Instructions:

1. Test the function with a large list (e.g., range(1000000)).
2. Use AI to: o Analyze time complexity. o Suggest performance improvements (e.g., using built-in functions, vectorization with NumPy if applicable). o Provide an optimized version.
3. Compare execution time before and after optimization.
4. Discuss trade-offs between readability and performance. Expected Output: An optimized function, such as: def sum_of_squares_optimized(numbers): return sum(x * x for x in numbers)

   Code

Testing the Original Function with a Large Input

Original Code

```
def sum_of_squares(numbers):
    total = 0
    for num in numbers:
        total += num ** 2
    return total
```

Test Input

```
numbers = range(1_000_000)
sum_of_squares(numbers)
```

✔️ The function produces the correct result but becomes **slow** for very large datasets due to Python loop overhead.

2. AI Analysis

Time Complexity Analysis

- The function iterates once over the list.
- **Time Complexity: O(n)**
- **Space Complexity: O(1)** (constant extra space)

Although the complexity is optimal, **Python-level loops are slower** compared to built-in or vectorized operations.

AI-Suggested Performance Improvements

Using AI tools such as **ChatGPT** or performance-aware linters, the following optimizations are suggested:

1. Replace explicit loops with **Python built-in functions**
2. Use **generator expressions** for cleaner and faster execution
3. (Optional) Use **NumPy** for very large numerical datasets

Optimized Version (Pythonic Approach)

```
def sum_of_squares_optimized(numbers):
    return sum(x * x for x in numbers)
```

✔ This avoids manual accumulation and leverages Python's optimized `sum()` function.

Optional: NumPy Vectorized Version (Advanced Optimization)

```
import numpy as np

def sum_of_squares_numpy(numbers):
    arr = np.array(numbers)
    return np.sum(arr * arr)
```

✔ Best suited for **very large datasets** and numerical workloads.

3. Execution Time Comparison

Version	Approximate Performance
Original loop-based version	Slowest
Generator + `sum()`	Faster
NumPy vectorized version	Fastest (for large arrays)

✔ All versions maintain **O(n)** complexity, but optimized versions reduce constant-time overhead.

4. Trade-offs: Readability vs Performance

Aspect	Readability	Performance
Original function	Very clear	Slower
Optimized Python version	Clear & concise	Faster
NumPy version	Less readable	Highest performance

- For general-purpose code, the optimized Python version is ideal.
- For data-heavy or scientific applications, NumPy is preferable.

Final Optimized Output (Expected Answer)

```
def sum_of_squares_optimized(numbers):
    return sum(x * x for x in numbers)
```

Conclusion

AI-assisted performance optimization helps developers identify bottlenecks even in logically correct code. By replacing explicit loops with optimized built-in functions or vectorized operations, execution speed can be significantly improved without changing algorithmic complexity. This approach is especially valuable in large-scale applications where performance and maintainability must coexist.